Continuous Monitoring of Nearest Neighbors on Land Surface

Songhua Xing
Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781
sxing@usc.edu

Cyrus Shahabi
Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781
shahabi@usc.edu

Bei Pan
Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781
beipan@usc.edu

ABSTRACT

As geo-realistic rendering of land surfaces is becoming commonplace in geographical information systems (GIS), games and online Earth visualization platforms, a new type of *k* Nearest Neighbor (kNN) queries, "surface" *k* Nearest Neighbor (skNN) queries, has emerged and been investigated recently, which extends the traditional kNN queries to a constrained third dimension (i.e., land surface). All existing techniques, however, assume a static environment, limiting their utility in emerging applications (e.g., Location-based Services) where objects move. In this paper, for the first time, we propose two exact methods that can continuously answer skNN queries in a highly dynamic environment which allows for arbitrary movements of data objects. The first method, inspired by the existing techniques in monitoring kNN in road networks [7] maintains an analogous counterpart of the Dijkstra Expansion Tree on land surface, called Surface Expansion Tree (SE-Tree). However, we show the concept of expansion tree for land surface does not work as SE-tree suffers from intrinsic defects: it is fat and short, and hence does not improve the query efficiency. Therefore, we propose a superior approach that partitions SE-Tree into hierarchical chunks of pre-computed surface distances, called Angular Surface Index Tree (ASI-Tree). Unlike SE-tree, ASI-Tree is a well balanced thin and tall tree. With ASI-Tree, we can continuously monitor skNN queries efficiently with low CPU and I/O overheads by both speeding up the surface shortest path computations and localizing the searches. We experimentally verify the applicability and evaluate the efficiency of the proposed methods with both real world and synthetic data sets. ASI-Tree consistently and significantly outperforms SE-Tree in all cases.

1. INTRODUCTION

With advances in remote sensing (e.g., LIDAR sensors) and computer graphics, a realistic, accurate and detailed rendering of Earth surfaces is becoming feasible in many applications. For example, both Microsoft Virtual Earth™ and Google Earth™ have started the display of accurate terrain models. Computer games, GIS systems and training simulators are other examples in which geo-realistic rendering of surfaces is included. Since the terrain models are no longer sparse nor based on synthetically generated data, disk-based structures are needed to store the large real-world datasets. Unfortunately, most data structures are designed to expedite the rendering of this geo-realistic data rather than its querying and access. The database community has recently started paying attention to this important but untapped area by studying a new type of *k* Nearest Neighbor (kNN) queries on surfaces, called *surface kNN* (*skNN*) queries [2, 3, 4].

Given a query point, a conventional kNN query [14] returns the number of *k* objects with the minimum distance with reference to this query point. In the case of skNN, the distance is measured by the surface distance. Note that this is different from 3D Euclidean space as the 3rd dimension is constrained by the terrain model. The skNN problem is analogues in some sense to supporting kNN query on road networks, where the distance is the network distance. However, the main difference is that the surface model, represented as triangular meshes, is much larger than road networks and even worse, the state-of-the-art algorithm to compute the surface distance (i.e., Chen-Han algorithm [1]) is much more complex than the Dijkstra algorithm [5] to find the shortest network path. To illustrate, note that the digital surface is usually represented as the Triangular Irregular Network model (TIN), a mesh generated from the sampled ground positions with 3D coordinates known as Digital Elevation Model (DEM). If we consider this TIN model as a graph with triangles' sides and vertices as edges and nodes, a reasonable size area (e.g., downtown Los Angeles) of 14km×10.7km using a 10m sampling interval contains about 1,498,000 nodes [11]. However, the graph of the road networks of the same area contains only 79,800 nodes [20]. Besides, the Chen-Han algorithm needs $O(N^2)$ time to compute the shortest surface path between a pair of points, where N is the size of the surface model. In [2], it is reported that this operation will take tens of minutes on a modern PC while it only takes a few seconds for the Dijkstra algorithm to compute the shortest path on a road network with the same size.

In this paper, for the first time, we study the problem of "Continuous" Surface *k* Nearest Neighbor (CskNN) query, which similar to its counterparts in Euclidean and network spaces, monitors the moving *k* nearest neighbors on land surfaces. Current skNN techniques [4] cannot support continuous queries efficiently as the data structures are expensive to update. Meanwhile, current continuous kNN methods on Euclidean and road networks cannot support CskNN due to the complexities of land surfaces.

In addition to their general utility in virtual environments, computer games, GIS and simulations, CskNN queries are applicable to a wide range of specific real-world applications as with CskNN the objects can arbitrarily move on land surfaces. For

VLDB '09, August 24-28, 2009, Lyon, France.

example, in the domain of disaster response, when one of the deadliest earthquakes hit Sichuan, a mountainous region in China, in May 2008, the entire transportation system collapsed, limiting movements to land surface. Under this circumstance, CskNN queries become extraordinarily important to save lives as they can monitor and coordinate among the moving objects (e.g., rescue teams) as well as provide exact shortest surface paths for evacuation purposes. In scientific research and environment protection domains, CskNN can be used to monitor the movements of animals to understand their habitats and the relationship between species. Finally, with recent advances in robotics, autonomy and ubiquitous sensors, new generation of terrain rovers are capable to perform difficult tasks in rough terrains for research in natural science and resource (e.g., coal, oil and mine) and space exploration. Therefore, it is essential to monitor these robots in order to maximize their effectiveness.

To tackle the CskNN queries, we start by studying the applicability of one of the most successful CkNN approaches in road networks to our problem. In particular, we introduce a counterpart of Dijkstra Expansion Tree [7] for surface, called Surface Expansion Tree or SE-Tree for short. For a static query point (or site), an SE-Tree computes the shortest surface path from the site to all the vertices of TIN. We show that unlike its counterpart, due to the special characteristics of land surfaces, SE-Tree is usually fat and short (e.g., for SE-Tree of a sample area with 11,406 vertices, the fan-out of the root node is 6,038) while the average fan-out of a node in Dijkstra Expansion Tree is usually small as a typical intersection (vertex) in a road network is a junction of two crossing roads. Nevertheless, studying SE-Tree and its features helped us to devise a more effective spatial index structure called Angular Surface Index Tree (ASI-Tree).

We build one ASI-Tree for each static query point (or site) as follows. First, centered at the query point, we partition the surface horizontally and vertically into several chunks. Each vertical partition, called Surface Shortest Path Container, includes vertical cuts on overlays of SE-tree. This is feasible due to the observation from the SE-tree structure that the shortest paths do not cross each other. The horizontal partitions are created by several contours centered at the query point, each of which with equal surface distance to the query point, called Surface Equidistance Lines. The intersection of these two partitions is called a *surface chunk*. Each surface chunk also stores some pre-computed information that can be used to expedite the exact shortest surface path computation. Next, an ASI-tree is built by hierarchically indexing these chunks rather than all the vertices in the triangular mesh.

ASI-Tree has several useful features. First, ASI-Tree is much smaller than SE-Tree (e.g., in the area of downtown Los Angeles, ASI-Tree only contains about 45,000 nodes). Second, since each container (a vertical partition) includes some pre-computed intermediate results (image sources), the time complexity of surface shortest path computation between the site and a moving object is reduced to $O(N\log(M)/M)$ where N is the size of the surface model and M is the total number of containers. This time complexity is even less than the Dijkstra algorithm ($O(N\log(N))$) if we consider the mesh as a network. Third, ASI-Tree is a well balanced thin and tall tree. The fan-out of the root is usually less than 8 (number of opposite edges of the query point) and the fan-out of the intermediate nodes is fixed (e.g., 2). Finally, ASI-Tree shrinks the search area within the small chunks containing the target moving objects. Our extensive experimental evaluation shows that ASI-Tree always outperforms SE-Tree in both I/O and CPU time by large margins.

The remainder of the paper is organized as follows. Section 2 briefly discusses some related research. In Section 3, we define the problem and provide some preliminaries. Section 4 describes a naïve CskNN algorithm based on the utilization of SE-Tree. Section 5 describes the details of our surface index (ASI) and its corresponding CskNN algorithm. In Section 6, we report the results of our experiments. Finally, in Section 7, we summarize the paper and discuss our future work.

2. RELATED WORK

In this section, we briefly survey the related work in the areas of kNN query processing, which can be classified into two categories of static and dynamic.

The first category is called static (or snapshot) queries. With this category, the query points, data objects and the underlying constrained environment (e.g., road networks, surfaces) are assumed as static during the query time. In the Euclidean space, Roussopoulos et al. [14] propose an R-tree based branch-and-bound kNN algorithm; Korn et al. [15] and Tao et al. [16] study the reverse nearest neighbor (RNN) problem, which finds the objects that take the query point as one of their nearest neighbors. Similar studies have been conducted in the constrained space (e.g., road networks, land surfaces). Papadias et al. [10] employ the Incremental Network Expansion (INE) algorithm to answer kNN queries in road networks. This algorithm performs a Dijkstra style expansion from the query point and examines objects in the order they are encountered. Kolahdouzan et al. [20] propose a Voronoi-based algorithm, VN^3, for spatial network databases. Samet et al. [9] present an algorithm which takes advantages of pre-computed shortest paths on the graph of the road networks. Recently, static kNN queries on land surface have been investigated as well. In [2, 3], Deng et al. propose a distance ranking method for the skNN query on the multi-resolution terrain model. However, since the exact shortest surface path is neither computed nor stored, continuous queries can only be answered as independent snapshot queries from scratch. Shahabi et al. [4] propose a surface R-Tree (SIR-Tree) based method which utilizes two surface indices (TSI and LSI) to process skNN queries efficiently and accurately. Unfortunately, neither TSI nor LSI can be applied to the scenario where objects are moving because TSI and LSI are built on static objects and online updating is costly.

With the second category, during the life of the query, the query points or objects or both are dynamic. In Euclidean space, Tao et al. [17] discuss time-parameterized queries, assuming linear trajectories for both the query point and objects. Tao et al. [18] also study the continuous NN (CNN) query which allows for arbitrary query movements while the objects are static, hence the input is a polyline which represents the query trajectory rather than a single query point. Yu et al. [21] propose two grid based algorithms to monitor nearest moving objects. Meanwhile, in road networks, Shahabi et al. [19] propose an embedding technique to transfer the road network to a constraint-free high dimensional space to fast but approximately retrieve nearest moving objects. Mouratidis et al. [7] propose an exact monitoring method for moving objects. In this method, a Dijkstra expansion tree is generated on the graph of the road network to keep track of the

shortest paths to all the nearest objects. This expansion tree grows or shrinks as the objects move away from or towards the query point. This paper considers three types of updates: object movements, query movements and fluctuations of edge weights. However, this method is more efficient to deal with object movements than the latter two, since both query movements and edge updates require online modifications or even the reconstruction of the entire expansion tree, which is usually expensive. To the best of our knowledge, there is no existing technique for any type of dynamic kNN queries on surface.

3. PRELIMINARIES

Before we explain the approaches for continuous monitoring algorithms, we would like to first state the underlying assumptions and formally define the problem. Next we explain a technique to compute the shortest path on surface and subsequently introduce the concept of *Surface Expansion Tree*.

3.1 Assumptions and Problem Definition

We assume a land surface is represented by the Triangular Irregular Network (TIN) model. A land surface also contains a set of moving objects and fixed position CskNN queries. A moving object is also called a point of interest. A CskNN query represents a request from a user at a fixed position to monitor its k closest objects. The query point is regarded as static during the query time, which could be a field observation station or a watch tower in real applications, while the objects move arbitrarily on the surface, which could be GPS attached vehicles or animals in real applications. Whenever the moving objects change locations, they send update requests to a centralized server to notify their new locations. Note that the objects or the query points are not necessarily located on the vertices of the TIN model.

Three distance metrics are typically considered on land surface: *Euclidean Distance* D_E, *Network Distance* D_N and *Surface Distance* D_S. The surface distance between two points on a terrain T is defined as the length of the shortest path connecting these two points on T while the network distance is defined as the length of the shortest path between two points on the graph G of T. In [4], the formal definitions of these three distance metrics and their relationships are provided. In sum, the Euclidean distance is always the lower bound of the surface distance while the network distance is always the upper bound of the surface distance.

Now we are ready to formally define the CskNN problem:

Problem Definition: Let T be the surface model and P be a set of moving data objects, given a query point q and a time interval τ, a CskNN query continuously identifies the k nearest objects in P to q based on the surface distance on T during τ.

Evaluating a CskNN query consists of two steps: 1) initial result computation as a snapshot skNN query which has been studied in [2, 3, 4]; and 2) continuously monitoring and updating the result sets as the objects move. The focus of this paper is on the latter.

3.2 Shortest Surface Path Computation

Chen-Han algorithm [1] is the state-of-the-art algorithm and has been widely used to compute the shortest path on a polyhedron surface. The basic idea of this algorithm is to first unfold all the faces of the polyhedron to one plane and then the straight lines on this plane that connect the source point to each vertex are the shortest surface paths from the source to these vertices. This algorithm costs $O(N^2)$ time and $\Theta(N)$ space, where N is the number of polyhedron faces. Chen-Han algorithm is practically expensive because there are many possible unfolding alternatives at each step of the algorithm. During the unfolding process, an unfolding sequence tree is kept in memory to traverse all the unfolding possibilities. This algorithm terminates when all the faces on the polyhedron surface have been processed (unfolded). We explain this algorithm using the following example [4].

Example 1: Figure 1 shows the process of computing the surface path between A and B on a tetrahedron. The triangular face 1, 2, 3 and 4 are unfolded to a plane with different unfolding alternatives (Case 1--3). The surface distance is the length of the shortest straight lines connecting A and B across all possible cases. The algorithm will compare the unfolding results and output Case 2 as the shortest surface path.

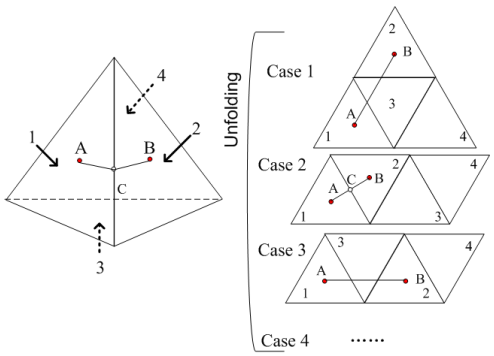


Figure 1. The unfolding process of Chen-Han Algorithm [4]

3.3 Surface Expansion Tree

With road networks, the *Dijkstra Expansion tree* is used to process kNN queries [7, 10]. This tree is generated by running Dijkstra Algorithm on the graph of a road network. The advantage of this tree is that it is embedded with the spatial proximity information of the underlying environment and is independent from the distribution of the data objects. Analogous to the Dijkstra Expansion tree, based on the Chen-Han algorithm, we define *surface expansion tree* as follows:

Definition 1 (Surface Expansion Tree): Let T be a surface model with the vertex set V and a source point s, the Surface Expansion Tree (SE-Tree) of T rooted at s is defined as the layout of Chen-Han algorithm, whose nodes are V and edges represent the shortest surface paths from s to these nodes.

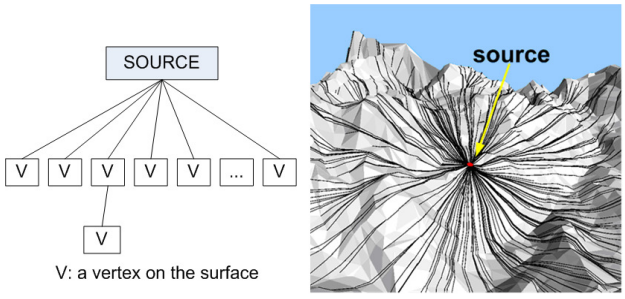


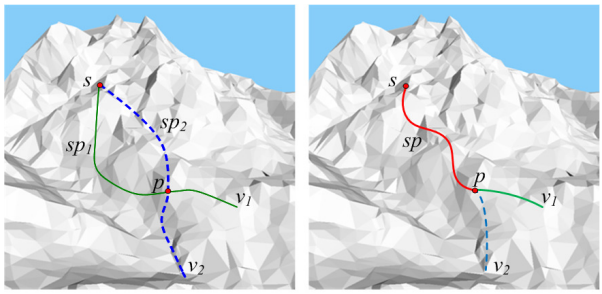
Figure 2. The SE-Tree and its overlay on surface

Figure 2 depicts the appearance of a Surface Expansion Tree. It is important to note the Surface Expansion Tree is different from the unfolding sequence tree since the Surface Expansion Tree is the final result of Chen-Han algorithm and there is only one path from the source to one vertex while the unfolding sequence tree is an intermediate result and all alternative paths are preserved.

From Figure 2, we have the following observations:

Observation 1: On a surface T with a source point s, any two shortest path sv_1, sv_2 from s to two different vertices v_1, v_2 do not cross each other.

Proof (by contradiction): Suppose sv_1, sv_2 cross at point p; thus sp_1 and sp_2 become two different paths from s to p as depicted in Figure 3(a). Without loss of generality, assume the length of sp_1 is equal or smaller than the length of sp_2, then the sum of the lengths of the paths sp_1 and pv_2 is equal or smaller than the length of sv_2, therefore the shortest path from s to v_2 is not sv_2, which contradicts our assumption. □



(a) Two shortest paths cross at one point (not possible)

(b) Two shortest paths share a common segments (possible)

Figure 3. Observation 1

However, it is possible but infrequent for different shortest paths to share common segments from the source point as Figure 3(b) depicts. If the *split point p* happens to be a vertex on the surface, then v_1, v_2 are p's descendants in the SE-Tree. Observation 1 is very important. It makes partitioning these surface shortest paths of an SE-Tree possible. Since these paths share a common source and do not cross each other, it is possible to partition the entire surface into several sector-shaped sub areas without cutting through any path. In Section 5, we will show how to utilize this observation to prune the search space.

Observation 2: Given a surface T and a source point s, the surface expansion tree of s has a very large fan-out at the root and a very low depth (i.e., a short and fat tree).

As depicted in Figure 2, unlike Dijkstra Expansion Tree on the road network, SE-Tree in general is fat and short because two paths rarely share any common segment. This observation points out the major drawback of SE-Tree: its extremely large size which is quadratic to the terrain size with an almost linear search time due to the tree being short and fat. Therefore, in Section 4, we only keep a very small fraction of the tree in memory at the just enough level to monitor the nearby data objects.

4. A NAIVE APPROACH

In this section, we describe a naïve approach which is a variation of techniques proposed in [7, 10] on road networks to surface by exploiting the counterpart of Dijkstra Expansion Tree of road networks, which is Surface Expansion Tree. We first present the algorithm and then analyze its drawbacks and deficiencies.

4.1 Initial Query Processing

We use the techniques based on MR3 framework [2] for initial query processing. We first describe the algorithm and then define the concepts of two areas used in this algorithm.

First of all, a 2D kNN Query is issued using Euclidean distance and k objects are acquired. Next, in the filter step, among these k objects, we select the one with the largest network distance and use this distance denoted as D_L to bound the search area as a range query in Euclidean Space. Finally, in the refinement step, we calculate the surface distances of all objects within this area and rank them to find the k nearest neighbors. In the result set, we denote the surface distance of the kth nearest neighbor as D_R.

Now we formally define *Expansion Area* and *Result Area*:

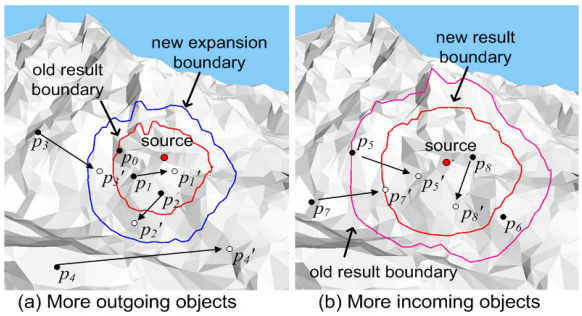
Definition 2 (Expansion Area): Let T be a surface model with a source point s, expansion area $EA(s)$ is a polygonal area around s, defined by $EA(s) = \{p: p \in T \text{ and } D_E(p, s) \leq D_L\}$. The boundary of $EA(s)$ is called the expansion boundary and denoted as $EB(s)$.

Definition 3 (Result Area): Let T be a surface model with a source point s, result area $RA(s)$ is a polygonal area around s, defined by $RA(s) = \{p: p \in T \text{ and } D_S(p, s) \leq D_R\}$. The boundary of $RA(s)$ is called the result boundary and denoted as $RB(s)$.

The expansion boundary bounds the area that needs unfolding by Chen-Han algorithm while the result boundary keeps track of the current result.

4.2 Object Movements

The data objects update their locations as they move. In this section, without loss of generality and to simplify discussion, we assume that there is only one query q in the system. Similar to [7], we classify the object movements into three categories: the movement within the result boundary, the incoming movement and the outgoing movement. We ignore those object movements that are outside the result boundary because they have no impact on the result set at all. Although movements within the result boundary may alter the ordering among k objects, the result set remains the same and we can ignore this case as well except the movement of the kth object where the result boundary itself needs updating. With respect to the number of objects passing through the result boundary, the maintenance of the result set can be divided into two scenarios depending on whether there are more outgoing objects or more incoming objects.



(a) More outgoing objects

(b) More incoming objects

Figure 4. Objects updates for 3-NN monitoring

If there are more outgoing objects than the incoming ones, the number of objects inside the original result boundary will be less than k and SE-Tree needs growing. As the example depicted in Figure 4(a) shows, both p_2' and p_3' become the candidates for kth NN since they are now in the area between the original result boundary and the new expansion boundary. On the other hand, if the outgoing objects are no more than the incoming ones, there are at least k objects located in the original result boundary and the result boundary probably shrinks. As the example depicted in Figure 4(b) shows, p_6 no longer belongs to the result set as it is outside of the new result boundary. Figure 5 depicts the algorithm to process a CskNN query.

Algorithm 1: CskNN I (surface T, query q, int k)
1. initialize min-heap H, stack *result*;
2. initialize SE-Tree t rooted at q;
3. *result* ←*Initial Processing* (T, q, k); // Section 4.1
4. calculate $RB(q)$;
5. if there is an update with $RB(q)$, let the number of outgoing object points as m and the number of incoming object points as n;
6. if $m \leq n$
7. update *result*;
8. shrink t and update $RB(q)$;
9. else
10. clear H and *result*;
11. add the object points inside $RB(q)$ to H
12. calculate $EB(q)$;
13. expand t within $EB(q)$;
14. for each object point p between $RB(q)$ and $EB(q)$
15. retrieve the surface distance $Ds(p, q)$;
16. add p to H;
17. while(*result.size* < k)
18. p' ←deheap H;
19. add p' to *result*;
20. update $RB(q)$;
21. keep monitoring until termination condition

Figure 5. The naïve algorithm for CskNN

Proposition 1: Let N be the size of the surface model T in number of vertices and m be the total number of objects, Algorithm 1's time complexity is $O(N^2 + m\log(m))$.

Proof: The major time consuming step is the expansion step (Line 12-13) where the surface distances are computed. The algorithm needs $O(N\log(N))$ time to calculate the expansion boundary and $O(N^2)$ time to compute the surface distance by using Chen-Han algorithm. Besides, it takes $m\log(m)$ to update results (Line 7, 14-19). Thus the total time complexity is $O(N^2 + m\log(m))$. □

Since $O(N^2)$ is usually at several orders of magnitude larger than $m\log(m)$, the overall time complexity of Algorithm 1 is $O(N^2)$. However, in the shrinking phase, there is no surface distance computation; hence the overall complexity is reduced to $m\log(m)$. Therefore, in our implementation, we cache the result of surface distance computation during the growing phase to avoid redundant computation for future object movements.

4.3 Analysis

Compared with techniques for kNN on road networks, this naïve approach shares some fundamental similarities. First, all these methods built an expansion tree rooted at the query point. Second, during the continuous monitoring phase, whether to update the result is determined by the result boundary (termed *Influencing Interval* in [7]). However, unlike Dijkstra algorithm, Chen-Han algorithm is suboptimal, that is, it is possible for SE-Tree to reach a vertex that has larger surface distance earlier than another vertex with smaller surface distance. Consequently, we need both the result boundary (to keep track of results) and expansion boundary (to bound the search) while these two boundary are the same on road networks.

This naïve approach could be fast during the phase when the SE-Tree shrinks. On the other hand, if there are more outgoing objects, the expansion tree has to grow to include k desired objects. In the case where there are many more outgoing objects or the kth nearest neighbor is far away, this algorithm may take tremendous CPU time for expansion processing. There are two reasons for this: 1) the overhead of online surface path computation is extremely high as shown in Proposition 1; and 2) the expansion area of SE-tree could be large. As shown in Figure 4(a), the expansion area from the old result boundary to the new expansion boundary is a ring-shaped area. To overcome these two problems, in Section 5, we create a data structure called *Surface Shortest Path Container* to store partial results of pre-computation and build a novel index schema called Angular Surface Index (ASI) which overlays the fat and short SE-Tree with another thin and tall tree in order to enhance the query efficiency.

5. SURFACE INDEX BASED ALGORITHM

In this section, we introduce a novel spatial index structure called *Angular Surface Index* (*ASI*) and its corresponding *ASI-Tree* which is a thin and tall tree structure to replace the fat and short surface expansion tree. Please note, the thin and tall tree here refers to a well balanced tree with a small branching factor (e.g., kd-Tree, Quad Tree). Towards this end, in Sections 5.1 and 5.2, we first introduce the concepts of two data structures: *Surface Shortest Path Container* and *Surface Equidistant Line* respectively, and then in Section 5.3 we present ASI and ASI-Tree which are built on top of these two data structures. In Section 5.4, we explain how this index structure can both localize the search and speed up the shortest surface path computation.

5.1 Surface Shortest Path Container

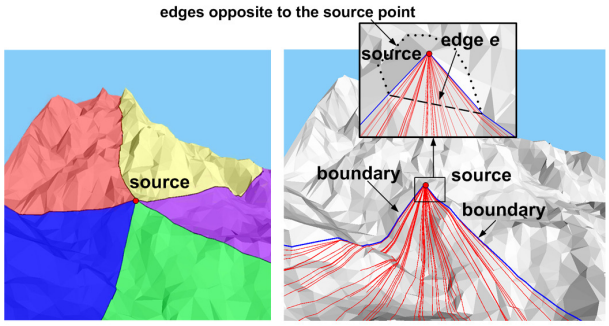
The simplest way to address the first drawback of the naïve approach is to pre-compute a complete SE-Tree offline and to store its corresponding shortest paths from the source. This approach could greatly speed up the online query. The shortest path computation could be divided into the following two steps: 1) locate the data object using a spatial index and 2) retrieve the shortest path directly from disk. However, this approach suffers from the following drawbacks: 1) in the case where a data object lays on the face rather than a vertex, this approach cannot find the exact shortest path and the accurate distance; 2) given a surface with N vertices, the space complexity of storing all these shortest paths is $O(N^2)$ per site, which is prohibitive especially since N could be in the order of millions and 3) the search time is almost linear. Therefore, in order to obtain precise results for arbitrary object movements and reduce both the space complexity and search time, we instead store geometric objects called *containers*. The goal of using *containers* is to take advantage of partial results based on geometric property to speed up the online process.

5.1.1 *Definition and Properties*

For road networks, the *shortest path container* is defined in [8]: Let $G = (V, E)$, w: $E \rightarrow R$ be a weighted graph, a set of nodes $C \subseteq V$ is called a *container*. A container C of an edge (u, v) is called *consistent*, if for all shortest paths from u to t that start with the edge (u, v), the target t is in C.

The advantage of using the shortest path container is to minimize the search area of Dijkstra algorithm. Similarly, if we can group the shortest paths on surface based on where they start from, we can prune the search space dramatically and speed up the query process. However, this classification technique on road networks cannot directly apply to surface because these surface shortest paths hardly share common segments and most containers would end up with only one vertex, which reflects the characteristic of SE-Tree: short and fat. Consequently, we propose a new concept of *Cover Set* and redefine the concept of *Shortest Path Container* for surface, and then discuss their spatial properties.

Definition 4 (Cover Set of an edge): Let T be a surface model with the vertex set V and a source point s. We call a set of vertices $CS \subseteq V$ on T a cover set of an edge e if CS consists of all and only the target vertices whose shortest paths from s intersect with e. We denote this cover set as $CS(e)$.



(a) Coloring different cover sets (b) a surface shortest path container

Figure 6. Cover Set and Container

The cover set of an edge defines a subset of vertices that can be reached by the shortest paths passing though that edge. To illustrate, we can use different colors to represent different cover sets of the edges opposite to the source point (Figure 6(b)). All the vertices of one cover set share the same color. Subsequently, the entire surface is divided into several disjoint regions. Figure 6(a) depicts this coloring. It is not surprising to find out that the vertices with the same color are close to each other. The following property determines its geometric boundary.

Property 1: Given the cover set $CS(e)$ of an edge e on a terrain T with a source point s, there must exist a geometric boundary b that encloses all and only the vertices of $CS(e)$ and their shortest paths.

The proof of Property 1 is straightforward and we omit the details here due to the space limit. Intuitively, as the shortest paths do not cross each other according to Observation 1, we can always find a polyline sp from the source s to a point p on the margin of T, which is immediately left to the leftmost shortest path to $CS(e)$ and do not cross any shortest paths, hence sp constitutes the left part of the boundary b. Similarly, the right part of the boundary b, sq also exists and the polyline on the margin between p and q, (p, q) constitutes the end boundary part. In some rare cases where none of the shortest paths to $CS(e)$ reaches the margin of T, we can choose p and q as the same point on the margin of T and the end boundary polyline (p, q) converges into one single point.

As a result of Property 1, for each cover set, a contour outline can be drawn as its geometric boundary. Now we give the formal definition of a container.

Definition 5 (Container of an edge): Let T be a surface model with the vertex set V and a source point s. A container of an edge e is defined as $C = (CS, B)$, where CS denotes the cover set of e and B is the geometric boundary of CS that satisfies Property 1. We denote this container as $C(e)$.

Figure 6(b) depicts a container $C(e)$ and all its shortest paths. Compared with the concept of container defined for road networks, the surface shortest path container is more rigorous since it does not include the vertices or paths of other containers. For each container, its boundary consists of: the left boundary line, the right boundary line and the end boundary line (which only exists if the left and right boundary lines do not converge). We will see how to draw this boundary in Section 5.1.2. Based on the boundaries, we can define the following relationships.

Definition 6 (Intersection): Let C_1 and C_2 denote two containers on a terrain T, C_1 intersect C_2 when their boundaries intersect.

Definition 7 (Contain): Let C_1 and C_2 denote two containers on a terrain T, C_1 contains C_2 when all the vertices of the cover set of C_2 are also in the cover set of C_1.

Definition 8 (Adjacency): Let C_1 and C_2 denote two containers on a terrain T, C_1 and C_2 are adjacent when they share a common boundary line.

Definition 8 is a little bit ambiguous since the boundary of a container is not unique because any curve separating two cover sets without crossing any shortest path can be viewed as a boundary line. Therefore, it is possible for two containers to be in fact adjacent without sharing any boundary line. However, we consider any two boundary lines as one as long as there is no vertex between these two.

Property 2: Let $C_1(e_1)$ and $C_2(e_2)$ denote two containers on a terrain T with a source point s, where neither C_1 contains C_2 nor C_2 contains C_1. C_1 and C_2 do not intersect if and only if e_1 and e_2 do not have any common shortest path from source s.

Proof: (by contradiction) Assume that C_1 and C_2 have no intersection, and e_1 and e_2 share one shortest path sp from the source s. Thus, sp's target point p must be included in both C_1 and C_2 according to the definition of container. In this case, since neither C_1 contains C_2 nor C_2 contains C_1, then C_1 and C_2 must intersect, which contradicts the assumption. □

Property 3: Let $C_1(e_1)$ and $C_2(e_2)$ denote two containers on a terrain T, C_1 contains C_2 if and only if all the shortest path from source s passing through e_2 also pass through e_1.

Property 4: Let $C_1(e_1)$ and $C_2(e_2)$ denote two containers on a terrain T, C_1 and C_2 are adjacent if and only if e_1 and e_2 are connected by a common vertex v and do not have any common shortest path from source s except through v.

(Proofs of Property 3 and 4 are similar to Proof of Property 2 and are hence omitted.)

Next, we define the container for a group of adjacent edges.

Definition 9 (Container of several adjacent edges): Let C_1, C_2, …, C_n denote n containers on a terrain T for n adjacent edges e_1, e_2, …, e_n. The container of these edges is defined as $C = (CS, B)$ where CS is the union of all cover sets of C_1, C_2, …, C_n, and B is the geometric boundary of CS that satisfies Property 1. We denote this container as $C(e_1,e_2, …, e_n)$.

In fact, the left and right boundary lines of $C(e_1, e_2 …e_n)$ can be drawn by tracing the outermost boundary lines of the two side containers and its end boundary line is the union of the end boundary lines of all the participating containers. In addition, the three operations (Intersection, Contain and Adjacency) for this type of container can be defined exactly the same as in Definitions 6, 7, 8 and Properties 2, 3 and 4 still hold. Similarly, the container of any arbitrary polyline on surface can be defined the same as Definitions 5 and 9 and all the above properties hold.

5.1.2 Creating Surface Shortest Path Container

In this section, we propose an algorithm to create a surface shortest path container.

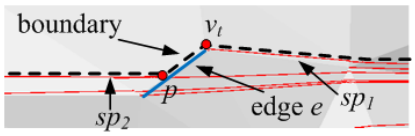


Figure 7. To trace left boundary line of a container

Algorithm 2: Trace Left Boundary (path *sp*)
1. initialize a stack B;
2. add *sp* to B;
3. let v be the destination vertex of *sp*;
4. while v is not on the margin of the surface
5. find the leftmost shortest path sp_2 (but right to *sp*) that intersects v's edges with an intersection p;
6. if (sp_2 is NULL) break;
7. add edge (v, p) to B;
8. add the path of sp_2 starting from p to B;
9. v ← destination vertex of sp_2;
10. *sp* ← sp_2;
11. return B;

Figure 8. The algorithm to trace the left boundary line

First of all, we sort shortest paths counter-clockwise where the target vertex of each path inherits the same sequence number as its shortest path. Figure 8 depicts the algorithm to trace the left boundary line of a container. Initially, the left boundary line is the container's leftmost shortest path sp_1 (assuming the source point s is on the right) as depicted in Figure 7. Once this path reaches its target vertex v_t and ends, this algorithm takes another shortest path sp_2 that intersects one of v_t's edges e and has the smallest (leftmost) sequence number after sp_1. We denote p as the intersection point of e and sp_2. In some very rare case, when none of v_t's edges intersects any other shortest path of this container, we expand edges from v_t in a Dijkstra style until the intersection point p is found. After adding edge (v_t, p), the left boundary line continues tracing sp_2. This process continues until the boundary line finally reaches a vertex on the margin of the surface or all the paths has been traversed. Figure 9 provides the complete algorithm to create a shortest path container. Please note, in Line 6, the end boundary can be NULL if left and right boundaries do not intersect the margin. The time complexity of Algorithm 3 is O(Nlog$N$) due to the sort operation in Line 3. However, since the pre-computation of shortest paths takes O(N^2), the overall time complexity is O(N^2).

Algorithm 3: Create A Container (polyline *l*)
1. initialize a container C;
2. *C.cover set* ← all the destination vertices of the shortest path intersecting *l*;
3. sort the shortest paths intersecting *l* counter-clockwise, let *sp* be the leftmost path and *sq* be the rightmost path;
4. *C.lBoundary* ←Trace Left Boundary(*sp*);
5. *C.rBoundary* ← Trace Right Boundary(*sq*);
6. *C.eBoundary* ← part of the margin of the surface that intersects its *C.lBoundary* and *C.rBoundary*;
7. return C;

Figure 9. The algorithm to create a container

5.2 Surface Equidistant Line

As shortest path containers partition a surface towards the vertical (longitude) direction, *Surface Equidistant Lines* are designed to partition along the horizontal (latitude) direction.

In cartography, a contour line which consists of points of equal elevation could be used to group the elevation information on a terrain. With the similar purpose, we can draw lines with equal surface distance to a fixed point.

Definition 10 (Surface Equidistant Line): Let T be a surface model with a source s. Given a distance value d, a surface equidistant line is defined as $l(s, d) = \{p: p \in T \text{ and } D_S(s, p) = d\}$.

For a given source point, several surface equidistant lines with different distance values divide the surface into disjoint rings. These lines are sorted by their increasing distance value to the source point and this order is termed as *levels* (e.g., the one with smallest distance is call 1st level surface equidistant line). These surface equidistant lines are represented as polylines on a surface model (see Figure 10(a)).

Property 5: Let l_1 and l_2 denote two adjacent polylines of a single equidistant line on a terrain T, there does not exist any shortest path from the source s that intersects both l_1 and l_2 unless at their common vertex v.

Proof: (by contradiction) Assume there is a shortest path *sp*, which intersects both l_1 and l_2 at p_1 and p_2, respectively. Without loss of generality, we assume that *sp* intersects l_1 first, so the length of the shortest path to p_2, is $|(s, p_2)| = |(s, p_1)| + |(p_1, p_2)|$. Since $|(p_1, p_2)| > 0$, it contradicts the definition of surface equidistant line that $|(s, p_2)| = |(s, p_1)|$. □

Combined with Property 4, we have the following corollary.

Corollary 1: Let l_1, l_2 denote two adjacent polylines of a surface equidistant line on a terrain T, then the two containers C_1 (l_1), C_2 (l_2) are adjacent.

5.3 Angular Surface Index

Based on surface shortest path containers and surface equidistant lines, a terrain surface can be divided into the following regions:

First, the surface is partitioned into m containers according to the m opposite edges of the source point. The opposite edges of a point p are all the edges of the triangles that have p as a vertex, but these edges do not share a vertex at p (see Figure 6(b)). For real world terrain models, m is usually smaller than 8. We call these containers *Primary Containers* or *1st Level Containers*. Second, inside each primary container, the region is partitioned by different surface equidistant lines. Next, in order to make partitions of equal size, we further continue partitioning according to each partition's proximity to the source point: each partition at level n will result in the number of b_f partitions at level n+1.To simplify our implementation, we fixed the branching factor b_f at 2 as depicted in Figure 10(a). We will further study the impact of b_f in our future work. For illustration purpose, a planar abstraction of ASI is depicted in Figure 10(b). Since each partition has an angular shape, this index structure is called **Angular Surface Index (ASI)** and each partition is called a *surface chunk*.

From the perspective of containers, ASI can also be viewed as a hierarchy of containers based on the "containing" relationship. Every surface equidistant polyline of each surface chunk is used to create a container. Inside one container, because the shortest paths that pass through one equidistant polyline l_1 at level n+1 must pass through another equidistant polyline l_2 at level n, as a result of Property 3, the container of l_1, $C_1(l_1)$ must be contained by the container of l_2, $C_2(l_2)$ and we say C_1 is a *child* of C_2. With ASI, each container has two children. According to Corollary 1, these two children are adjacent. Therefore, a tree is built for ASI construction, called **Angular Surface Index Tree (ASI-Tree)**.

Figure 10(c) depicts an ASI-Tree. With this ASI-Tree, each node represents a container. The root node is the source point which could be viewed as a super container that contains the entire surface. Except for the root node, all intermediate nodes have a fan-out of two. Besides pointers to its children, each intermediate node also stores pointers to two polylines: one polyline that separates its own chunk from its children, termed *chunk separator* and another polyline that separates its two children, termed *child separator*. The root usually has more than one *child separators*. To facilitate shortest path computation, each node also stores the image sources (defined in [1]) for the vertices in the chunk. An image source for a point p is the image of the source that is coplanar with the face containing p for a given unfolding and is an intermediate result while running Chen-Han algorithm. However, since many faces can be reached by one unfolding, the total number of image sources are much less than the number of faces (or vertices) inside this chunk. We refer readers to [1] for details. Unlike R-Tree [12] whose node size can be chosen to be equal to (or a multiple of) disk page size, ASI-Tree has a fixed node size, thus is stored on disk the same way as other external data structures with fixed node size such as Quadtree or K-D tree [13].

Compared with SE-Tree, ASI-Tree has the following advantages:

1. **Storage:** ASI-Tree has M nodes while SE-Tree has N nodes. (N is the terrain size while M is the total number of chunks).
2. **Regularity:** ASI-Tree is almost balanced while SE-Tree is very irregular and surface dependent.
3. **Efficiency:** ASI-Tree has a small fan-out for each node and the search time is logarithmic while SE-Tree is usually fat and short and the search time is almost linear.

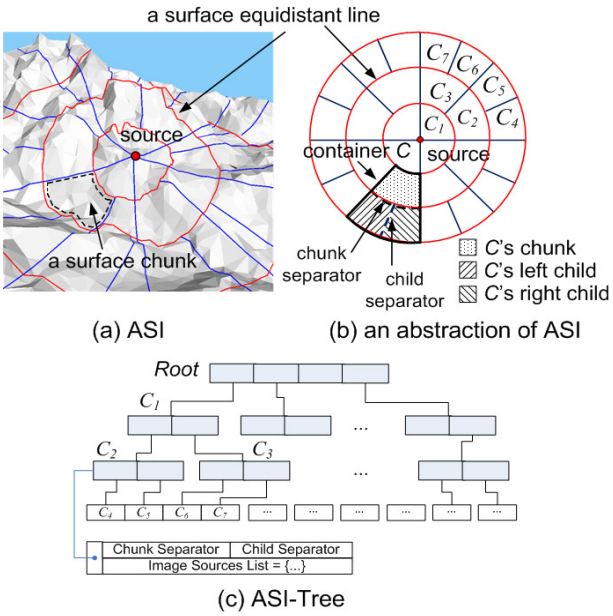


Figure 10. ASI and ASI-Tree

Please note, the ASI-tree for the entire terrain is usually not fully balanced as the query point is not always at the center of the terrain or in the special case where the query point is placed near a steep terrain surface, causing some containers not covering any vertex on the margin. Fortunately, this "unbalance" has little impact on our problem. This is because, kNN queries typically concern about the nearby objects and usually can be answered before reaching the leaf nodes of the ASI tree (the margin of the surface). Hence, for the majority of cases, the part of the ASI tree that will be visited is balanced and the search time is logarithmic.

Algorithm 4: Create All Containers (polyline l)
1. container C ← Create A Container(l); // Algorithm 3
2. if C does not contain any equidistant line // a leaf node
3. return C;
4. else
5. polyline l' ← the lowest level equidistant polyline in C;
6. divide l' into two equal length polylines l_1 and l_2;
7. C's left child ← Create All Containers(l_1);
8. C's right child ← Create All Containers(l_2);
9. return C;

Figure 11. The algorithm to create all containers

Algorithm 5: ASI Construction (surface T, source s, int n)
1. initialize an ASI-Tree *root* at s;
2. for each edge e_i opposite to s
3. primary container $PC(e_i)$ ← Create A Container(e_i);
4. *root.child[i]* ← $PC(e_i)$
5. create n equidistant lines, next, sort them by levels;
6. for each $PC(e_i)$
7. polyline l ← first equidistant polyline inside $PC(e_i)$;
8. divide l into two equal length polylines l_1 and l_2;
9. $PC(e_i)$'s left child ← Create All Containers(l_1);
10. $PC(e_i)$'s right child ← Create All Containers(l_2);
11. return *root*;

Figure 12. The algorithm for ASI construction

Figure 12 provides a recursive algorithm to construct an ASI-Tree. In Algorithm 5, n indicates the total number of equidistant lines. If the equidistant lines are selected at the constant distance interval, the total number of chunks will be exponential with the terrain size, which is prohibitive. Therefore, the equidistant lines should be selected at the distance multiplied by $2^{n/2}$, making the number of chunks linear with the terrain size. In experiments, n is selected automatically by a parameter called Container Density D_C that $n = \log(ND_C)$ where N is the terrain size.

Algorithm 6: Compute D_S (surface T, ASI-tree r, point p)
1. container *node* ← r;
2. while (*node* != NULL)
3. if p is on the upper region of *node.chunk separator* // this chunk contains point p
4. search the image source s' for p and compute $D_E(s', p)$;
5. return $D_E(s', p)$;
6. else
7. find number m that p is between *node*'s mth child separator and *node*'s $m+1$th child separator
8. *node* ← *node*'s mth child;

Figure 13. The algorithm for shortest path computation

Figure 13 sketches an algorithm to use the ASI-Tree to locate which chunk a data object falls in and compute its surface distance. In Line 4, upon identifying which chunk the object p falls in, its surface shortest distance is then computed based on the image sources stored for this chunk. According to the definition, the image source s' is coplanar with the face p locates, its surface distance can be computed immediately as the Euclidean distance between s' and p. In order to address kNN only, this process is enough. For cases where we also need the actual path, we need to unfold this chunk and all its ancestors. We apply a variation of Chen-Han algorithm where the image source is known in advance: first, connect the image source s' and p to form a straight line $s'p$, then start unfolding from p and always choose the face that overlaps with the line $s'p$. Apparently, this variation runs linearly with the number of vertices in the unfolding area.

The time complexity of Algorithm 6 is $O(N\log(M)/M)$ because the algorithm needs $O(N\log(M)/M)$ time to locate p and $O((N/M))$ time to find the image source and compute its surface distance. If an exact surface path is desired, the unfolding process costs $O(N\log(M)/M)$ but the overall complexity remains the same.

ASI can also be indexed by SIR-Tree proposed in [4]. For each surface chunk, a representative point can be selected for the R tree index. Correspondingly, each leaf node records a pointer to the vertex list of each chunk. However, this R-tree based approach does not take advantage of the hierarchical spatial containing relationship of containers and results in a larger search area.

5.4 CskNN Query Processing

In this section, we present our algorithm to process CskNN queries using ASI-Tree and explain why this algorithm will reduce the search area compared with the naïve approach. The core of our CskNN query processing is the same as discussed for the naïve approach in Section 4.1 and 4.2. The main difference is that we utilize ASI-Tree to localize the search area for candidates and reduce the complexity of surface shortest path computation.

Algorithm 7: Initial Processing (surface T, ASI-Tree r, query q, int k)
1. initialize min-heap H, stack *result*, candidate set Q, Q_2
2. Q ← kNN(q); // 2D k-NN query in Euclidean Space
3. for each object point p in Q
4. Compute $D_S(T, r, p)$; // Algorithm 6
5. add p to H;
6. D_{max} ← maximum surface distance of objects in Q;
7. Q ← Range(q, D_{max}); // range query in Euclidean space
8. find number m that there are $k_1 (\leq k)$objects inside the mth surface equidistance line and $k_2 (> k)$ objects within the $(m+1)$th surface equidistance line; then insert the objects within the $(m+1)$th surface equidistance line into Q_2
9. $Q \leftarrow Q \cap Q_2$
10. for each point object p in Q but not in H
11. Compute $D_S(T, r, p)$; // Algorithm 6
12. add p to H;
13. while(*result.size* < k)
14. p' ←deheap H;
15. add p' to *result*;
16. return *result*;

Figure 14. the algorithm for initial query processing

Figure 14 depicts the initial result computation phase. Similar to the naïve approach, Algorithm 7 shares the same filter and refinement framework as in Section 4.1. However, it has the following differences. First, in Line 4, during the first filtering step, the actual surface distances to candidates are computed using ASI-Tree rather than the network distances. This not only results in a smaller search area, but more importantly, makes the complexity of this process ($O(N\log(M)/M)$) even less than that of the Dijkstra algorithm ($O((N\log(N))$) if we consider the mesh as a network. Second, in Line 8, a second filtering step is employed which takes advantage of the property of surface equidistance lines, thus the size of the candidate set is reduced even further.

Algorithm 8: CskNN II (surface T, ASI-tree r, query q, int k)
1. initialize min-heap H, stack *result*, candidate set Q, Q_2
2. *result* ←Initial Processing (T, r, q, k) and calculate $RB(q)$; // Algorithm 7
3. if there is an update with $RB(q)$, let the number of outgoing object points as m and the number of incoming object points as n;
4. if $m \leq n$
5. update *result*;
6. update $RB(q)$;
7. else
8. clear H and *result*;
9. Q ←apply filter1 (T, r, q, k);
10. Q_2 ←apply filter2 (T, r, q, k);
11. add the object points inside Q_2 to H
12. $Q \leftarrow Q - Q_2$;
13. for each object point p in Q
14. compute $D_S(T, r, p)$; // Algorithm 6
15. add p to H;
16. while(*result.size* < k)
17. p' ←deheap H;
18. add $p'$to *result*;
19. update $RB(q)$;
20. keep monitoring until termination condition;

Figure 15. The surface index based algorithm for CskNN

In the continuously monitoring phase as depicted in Figure 15, a result boundary is used to check when an update is issued as in Section 4.2. In Algorithm 8, the first filter (upper bound) is applied in Line 9, which is exactly the same as Lines 2-9 in Algorithm 7. The second filter (lower bound) is applied in Line 10, which is similar to Line 8 in Algorithm 7, but only inserts the objects within the mth surface equidistance line into Q_2. Therefore, the candidate set for kth NN is the difference set of the first filtering result set and the second filtering result set (Line 12). Consequently, the search area is the union of the chunks of all the objects in this candidate set, which is much smaller than the search area of the naïve approach (the area between the old result boundary and the new expansion boundary).

Example 2: Figure 16 shows the comparison between the two proposed approaches. In this example to monitor 3NN, we assume that only p_3 is moving. We can see that the ASI-based approach reduces the search area remarkably by a more powerful filter (e.g., p_5 is filtered out from candidates) and localizing the search only inside the chunks of the candidates (e.g., p_3' and p_4).

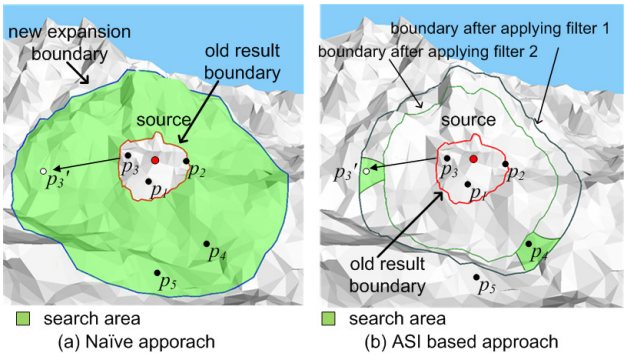


Figure 16. In comparison of two approaches, the search areas are shaded in green (dark grey in black white printing).

Proposition 2: Let N be the size of the surface model T, m be the total number of objects and M be the total number of chunks, Algorithm 8's time complexity is $O(mN\log(M)/M + m\log(m))$.

Proof: The major time consuming steps are applying filter 1 (Line 9) and the result updating (Line 5, 16-18). In applying filter 1, at most m times surface distance computation by using Algorithm 6 are needed, which is $O(N\log(M)/M)$, while the result updating is the same as in Algorithm 1 which is $m\log(m)$. Thus the total time complexity is $O(mN\log(M)/M + m\log(m))$. □

6. PERFORMANCE EVALUATION

6.1 Experimental Setup

In our extensive experiments, both large scale real-world and synthetic datasets are used. The real-world surfaces are modeled by the 10-m USGS DEM data sets downloaded from [11] which are the same as the datasets used in previous studies in [2] and [4]: 1) Bearhead (BH) area in WA, USA which covers an area around 10.7km×14km and 2) Eagle Peak (EP) area in WY, USA with similar size as BH. In addition, in order to analyze how the land surface itself can affect the performance of our proposed method, we create five synthetic surface models with the same size (10km×10km) where we can vary *roughness* R_A as a parameter. The *roughness* of a surface has different measures and we use the most common one: the standard deviation of the elevations away from the mean elevation of a scan line over the surface.

For all experiments, we use a PC with Intel 6420 Dual CPU 2.13G Hz and 3.50 GB RAM. The operating system is Windows XP SP2. To compute the exact surface distance, we use the most recent implementation of Chen-Han algorithm in [6], and all the algorithms are implemented in Microsoft Visual Studio 2005. In the implementation of Algorithm 6 (Figure 13), we choose the linear variation of Chen-Han algorithm; therefore, both the k nearest neighbors and shortest surface paths can be provided.

6.2 System Parameters

Parameters	Default	Range
Number of NN k	10	2, 4, 6, 8,…, 20
Object Distribution	Uniform	Uniform, Gaussian
Object Density D_O	0.6%	0.2, 0.4, 0.6, 0.8, 1 (%)
Object Agility a	10%	0, 5, 10, 15, 20 (%)
Object Speed v	30 m/ts	10, 20, 30, 40, 50 (m/ts)
Container Density D_C	3%	1, 2, 3, 4, 5 (%)
Surface Roughness R_A	10%	6, 8, 10, 12, 14 (%)

Table 1. System parameters

For our experiments, we conduct 100 CskNN queries to evaluate the average performance of the system under different values of parameters listed in Table 1. Each query requires continuous monitoring for 50 timestamps. For each set of experiments, we only vary one parameter and fix the remaining to the default values. The performance is measured by the average query processing time and I/O cost per timestamp. The first 6 parameters are tested on both BH and EP while Surface Roughness R_A is only tested on synthetic data sets. Initially, data objects follow either uniform distribution or Gaussian distribution (with mean at the center of the land surface and standard deviation of 10% of the maximum surface distance from the center) with varying densities from 0.2% to 1%. The object agility indicates the overall objects activeness, measured by the percentage of moving objects per timestamp while the object speed indicates each object's activeness, measured by meters per timestamp. Please note, since objects can move arbitrarily and do not necessarily follow the surface shortest path, it is impossible to use the exact path to compute the speed of every object and hence the speed is measured by the displacement in Euclidean space per timestamp. Container density is defined as the ratio of the number of containers (or chunks) over the terrain size. This parameter can also be used to estimate the ASI tree's size (in number of nodes).

6.3 Performance Study

Since this paper is the first to address the continuous kNN problem on surface, we compare the two approaches proposed in this paper as competitors.

6.3.1 The Impact of k

First, we compare the performance of two algorithms by varying the value of k from 2 to 20 on both BH and EP while using default settings (see Table 1) for all other parameters. Figure 17 shows the average query efficiency and number of I/O operations (as a function of the number of retrieved surface vertices). The result indicates that ASI based algorithm outperforms the naïve

algorithm both in query efficiency and I/O operations. As the value of k increases, the response time of the naïve algorithm grows at a quadratic rate due to the invocation of Chen-Han algorithm while the response time of ASI-based algorithm increases at a linear rate. ASI-based algorithm outperforms the naïve algorithm by at least a factor of two for k > 4. ASI-based algorithm also yields a better performance in I/O by an average factor of two because the search is localized to avoid unnecessary access to surface vertices.

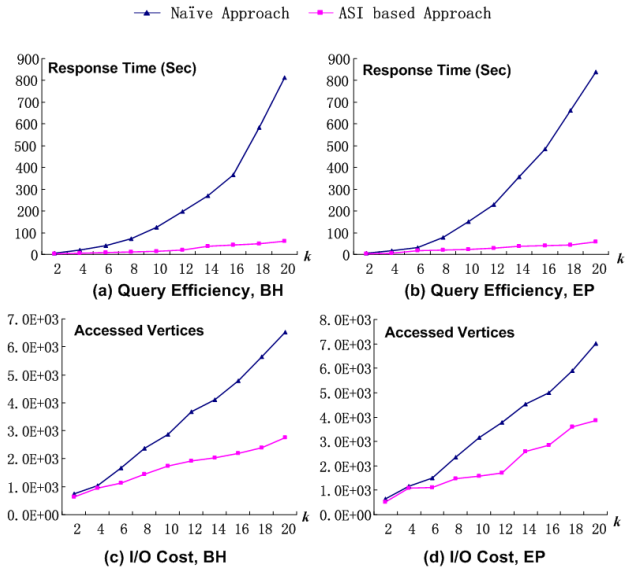


Figure 17. Query Efficiency, I/O Cost vs. Value of k

6.3.2 *The Impact of Object Distribution and D_O*

Next, we study the impact of initial object distribution. Figure 18 shows the query processing time and I/O cost on BH and EP with objects which follow either uniform or Gaussian distributions. Clearly, ASI-based algorithm outperforms the naïve algorithm significantly in all cases. In addition, the difference between these two distributions is not noticeable. In our experiments, the ASI based algorithm has a slightly better performance for objects with Gaussian distribution than objects with uniform distribution.

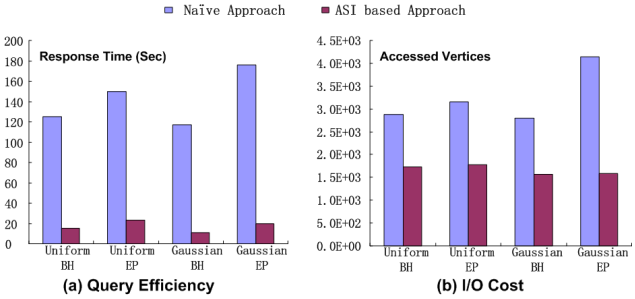


Figure 18. Query Efficiency, I/O Cost vs. Object Distribution

Figure 19 depicts the performance of the two proposed algorithm by varying the object density from 0.2% to 1%. In general, with a fixed value of k, both query processing time and I/O cost decrease for both algorithms as D_O increases. This is because a higher density usually results in a smaller search area, and the overhead for both algorithms is reduced. Furthermore, even at the highest density, ASI-based algorithm outperforms the naïve algorithm by a factor of five in query efficiency and a factor of two in I/O cost.

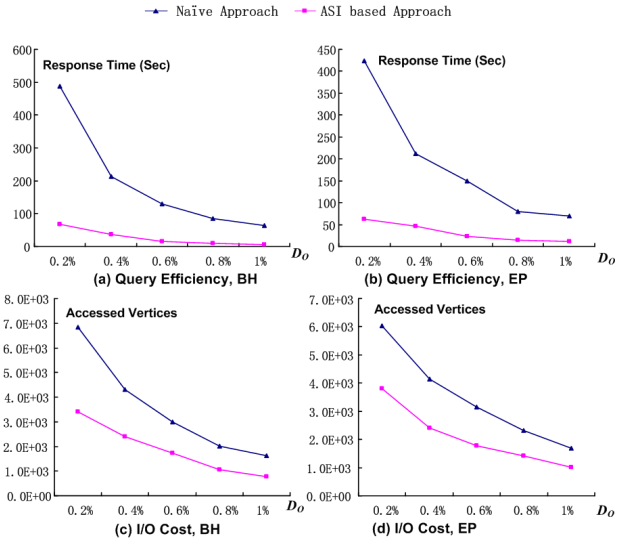


Figure 19. Query Efficiency, I/O Cost vs. D_O

6.3.3 *The Impact of a and v*

In the next set of experiments, we study how the object movements affect the performance. Two parameters are used to measure the movements, which are object agility a and object speed v. We first fix v to study the impact of a and then fix a to study the impact of v. All other parameters are fixed to the default settings (see Table 1).

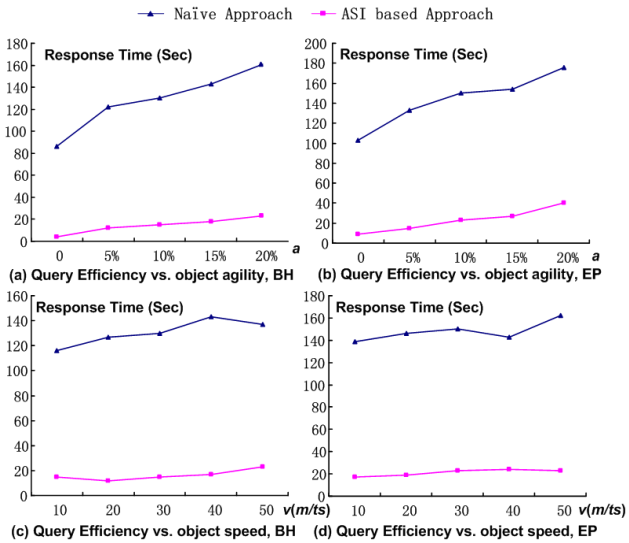


Figure 20. Query Efficiency vs. a & v

Figure 20 (a) and (b) illustrate the impact of object agility. As the object agility grows, both algorithms' query processing time increases slightly as well because the possibility to enlarge the search area is increased. However, as Figure 20 (c) and (d) illustrate, both algorithms are practically unaffected by object speed because the core of both algorithms only concern whether

there are object updates rather than how far the objects move. Clearly, ASI-based algorithm has an obvious predominance over the naïve algorithm in all cases. The trends of I/O cost for both algorithms are the same as query response time and hence omitted.

6.3.4 The Impact of D_C

Figure 21(a) depicts the impact of container density with respect to query efficiency. Only the performance of ASI based algorithm is evaluated as containers are not used in the naïve algorithm. We observe that the performance is enhanced as more containers are created for both BH and EP.

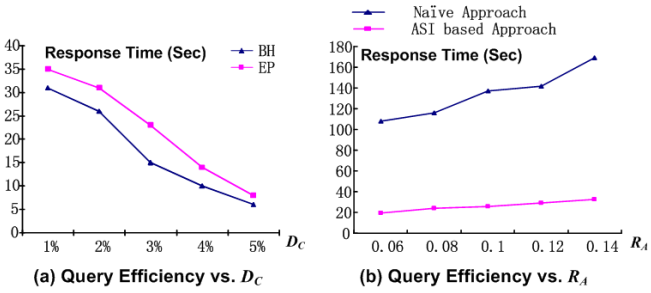


Figure 21. Query Efficiency vs. D_C & R_A

6.3.5 The Impact of R_A

In our final set of experiments, we vary the roughness R_A over five synthetic surface models of the same size. Regardless of the value of R_A, ASI-based algorithm keeps outperforming the naïve algorithm at a factor of five. In addition, as shown in Figure 21(b), rougher terrains incur more query processing time. This is reasonable because the differences among three distance measures (i.e., Euclidean distance, Network distance and Surface distance) are usually amplified with rougher terrains, thus the filters in both Algorithm 1 and Algorithm 8 could probably generate a larger search area than smooth terrains.

7. CONCLUSION AND FUTURE WORK

In this paper, for the first time, we introduce the continuous monitoring kNN problem on surface and propose two algorithms: a naïve algorithm which is analogous to the same problem in road networks and a surface index (ASI) based algorithm. The experiment results show that the ASI-based algorithm outperforms the naïve algorithm under all circumstances by a factor of 2 to 10.

Due to the complexity of dealing with land surfaces, in this paper we assume a simplified problem setting (pre-defined static query points) which in turn restricted our motivating real-world applications. However, our observations of the characteristics of shortest paths on land surfaces are fundamental and can be stepping stones for future research in this area addressing more complex settings and hence more real-world applications. Some extensions of CskNN problem with more complex settings are straightforward. For example, to relax the "pre-defined" query points to any arbitrary query point on surface, the ASI can be built for every vertex, pre-computing its surface distance to all other vertices. This combined ASI can answer arbitrary queries by disassociating the surface space from query points, very much like the work in recent literature (e.g., [9]) that utilizes this idea on the road network. Our future plan includes further studying these complex settings, where queries move arbitrarily.

8. ACKNOWLEDGMENTS

This research has been funded in part by NSF grants IIS-0238560 (PECASE), IIS-0534761 and CNS-0831505 (CyberTrust), the NSF Center for Embedded Networked Sensing (CCR-0120778) and in part from the METRANS Transportation Center, under grants from USDOT and Caltrans. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] J. Chen and Y. Han: Shortest paths on a polyhedron. 6th ACM Symp. Comput. Geometry, pages 360–369, 1990.

[2] K. Deng, X. Zhou, H. T. Shen, K. Xu, X. Lin: Surface k-NN Query Processing. ICDE 2006.

[3] K. Deng, X. Zhou, H.T. Shen, Q. Liu, K. Xu and X. Lin: A Multi-resolution Surface Distance Model for k-NN Query Processing. The VLDB Journal, Volume 17, August 2008.

[4] C. Shahabi, L. Tang and S. Xing: Indexing Land Surface for Efficient kNN Query. PVLDB Volume 1(1), 2008.

[5] E. W. Dijkstra: A note on two problems in connection with graphs. Numeriche Mathematik, 1:269–271, 1959.

[6] B. Kaneva and J.O'Rourke: An implementation of Chen & Han's shortest paths algorithm. Proc. of 12th Canadian Conf. on Comput. Geom, 2000.

[7] K. Mouratidis, M. Yiu, D. Papadias and N. Mamoulis: Continuous nearest neighbor monitoring in road networks. VLDB 2006.

[8] D. Wagner, T. Willhalm, C. Zaroliagis: Geometric containers for efficient shortest-path computation. J. Exp. Algorithmics

[9] H. Samet, J. Sankaranarayanan,H. Alborzi: Scalable network distance browsing in spatial databases. SIGMOD 2008.

[10] D. Papadias, J. Zhang, N. Mamoulis, Y. Tao: Query Processing in Spatial Network Databases. VLDB 2003.

[11] Http://data.geocomm.com.

[12] A. Guttman: R-trees: a Dynamic Index Structure for Spatial Searching: SIGMOD 1984.

[13] H. Samet: Foundations of Multidimensional and Metric Data Structures 2005.

[14] N. Roussopoulos, S. Kelley, and F. Vincent: Nearest neighbor queries. SIGMOD 1995.

[15] F. Korn, and S. Muthukrishnan: Influence Sets Based on Reverse Nearest Neighbor Queries. SIGMOD 2000.

[16] Y. Tao, D. Papadias, and X. Lian: Reverse kNN search in arbitrary dimensionality. VLDB 2004.

[17] Y. Tao and D. Papadias: Time-parameterized queries in spatio-temporal databases. SIGMOD 2002.

[18] Y. Tao, D. Papadias and Q. Shen: Continuous Nearest Neighbor Search. VLDB 2002.

[19] C. Shahabi, M. R. Kolahdouzan and M. Sharifzadeh: A road network embedding technique for k-nearest neighbor search in moving object databases. ACM-GIS 2002.

[20] M. Kolahdouzan and C. Shahabi: Voronoi-based k nearest neighbor search for spatial network databases. VLDB 2004.

[21] X. Yu, K. Q. Pu, and N. Koudas : Monitoring k-nearest neighbor queries over moving objects. ICDE, 2005.